TUNeEngine : An Adaptable Autonomic Administration System

Alain Tchana, Suzy Temate, Laurent Broto, Daniel Hagimont

TUNeEngine: An Adaptable Autonomic Administration System

Alain Tchana, Suzy Temate, Laurent Broto, Daniel Hagimont
Toulouse University, IRIT Laboratory
Toulouse, France
first.last@enseeiht.fr

***Abstract*—The Autonomic Administration technology has proved its efficiency for the administration of complex computing systems. However, experiments conducted with several Autonomic Administration Systems (AAS) revealed the need to adapt the AAS according to the administrated system or the considered administration facet. Consequently, users usually have to adapt even to re-implement the AAS according to their specific needs but these tasks require high expertise on the AAS implementation that users do not necessarily have. In this paper we propose a service-oriented components approach to build a generic, flexible, and useful AAS. We present an implementation of this approach, the design principles and the prototype called TUNeEngine. We illustrate the flexibility of this prototype through the administration of a complex computing system which is a virtualized cloud platform.**

***Keywords*-Autonomic Administration; Adaptable System; Components Model**

I. Introduction

These last decades, computer systems became increasingly sophisticated and thus more complex to manage. The autonomic administration technology, introduces in 2003 by IBM[1], has proved its efficiency to cope this complexity. It consists in the substitution of human administrators by computer programs, called Autonomic Administration Systems (or AAS for short)[1][2][3], in order to perform usual administration tasks while limiting as much as possible human interventions. Since the introduction of this technology, we observed an increasing number of AAS project which can be classified into two categories: specialized AAS and generic AAS. The former includes AASs which are developed for administrating a specific application (or type of applications) [5][6], or dedicated to a specific facet of administration (e.g. deployment) [7]. Regarding the second category, it includes AASs whose ambition is to handle any type of applications [8][9].

Our research team has developed two AASs, Jade [10] and TUNe [9], with the ambition to make them generic. Both TUNe and Jade are based on the Fractal component model. Jade required high expertise on Fractal and Java from the administrator, so TUNe was developed to raise the level of abstraction of Jade to make it more usable. Section II presents an overview of TUNe. We experimented TUNe with different types of applications: multi-tier (JEE) [9], large scale [19], and virtualized applications [11] and we noticed that TUNe was not as generic as we had liked it to be. We observed that when we move from one type of applications on to another one, we usually had to change TUNe deeply to take into account the new administrations needs. For example, virtual machines have the specificity to be software which behave like machines but TUNe was implemented in such a way that it differentiated machines from software and thus did not allow to take virtual machines duality into account. Therefore, to enable virtual machines administration, we changed the machines and the software behaviors in TUNe. The main problem is that this adaptation required the modification of the overall system and this is rarely accessible for an administrator whose expertise is out of the scope of developing an AAS. Based on our experiments, it emerges that this problem results from three points: (1) strong implementation hypothesis; (2) unsuitable design choices and (3) difficulty of use. Section II-B details each of these three points. This problematic is also observed on others AASs such as Rainbow [8], Unity [13] or Accord [14] (Section VI presents some research works on this topic).

In previous work [15], we proposed a model driven approach to address this problem. Briefly, this approach consists in using Domain Specific Languages (DSL) to express specific administration policies according to the considered type of applications. These specific policies must then be implemented by a higher flexible and adaptable AAS. Section II-C summarizes our general approach which is divided into two parts, the DSL description part and the generic AAS they have to rely on. This paper presents the second part of this approach: the implementation of a higher flexible AAS which can be adapted without having an expert knowledge of its implementation. More precisely, the contributions of this paper are:

- we propose a set of guidelines which can be seen as design principles that the development of a higher flexible and adaptable AAS can follow (presented in Section III).
- we propose a prototype of such a AAS whose implementation is based on a service-oriented components model (presented in Section IV).
- We evaluate the flexibility and the adaptability of our prototype over the administration of a complex environment: a virtualized cloud platform, including hosted applications (presented in Section V).

We conclude in section VII.

II. Background and Problematic

A. Background: the TUNe AAS

An autonomic administration system (AAS) is a system which is able to perform administration tasks (Deployment, configuration, repairing ...) in an automatic way with no need of human interventions. TUNe is an AAS which follows the Kephart's AAS [1] model, generally known in the literature under the term *MAPE-K* (Figure 1). The AAS monitors the runtime environment via probes. The latter inform the AAS (with notifications) about particular situations (e.g. machines failure). When the AAS decide to react to probes notifications, it performs some reconfiguration programs which affect the runtime environment through actuators. Among the existing AASs, TUNe is one of those which provide all of the administration services: wrapping (make a legacy element administrable by the AAS), deployment (make legacy files/binaries available in the runtime environment), and dynamic (re)configuration at runtime (capability to dynamically perform a set of actions in the runtime environment). In order to facilitate its use, TUNe provides high level languages to describe the mentioned administration services, in lieu of low level API generally provided in others AASs. TUNe's languages are based on UML (widely used by administrators) and each language is dedicated to an administration service:

Wrapping: as its ancestor Jade, TUNe is a component based AAS where each administrated element (hardware or software) is encapsulated (wrapped) in a component [16]. To encapsulate an element means to capture its properties and behaviours in order to provide a data structure (called component) which represents this element in the AAS. Then, the AAS lays on the component model's APIs to easily perform administration tasks. This approach has been proved in other AASs such as Rainbow [8] (developed by IBM). TUNe provides wrapping features through out two languages. The first one, called Architecture Description Language (ADL), is used to describe administrated elements, their properties and the relationships between them. This language uses the graphical UML class diagram, which is much more intuitive than a traditional ADL (which is XML like). The second language is textual and called Behavior Description Language (BDL). It is used to express the behavior of administrated elements.

Deployment: even if the same language is used for wrapping software or wrapping machines, TUNe differentiates the wrapping of software elements from wrapping of infrastructure elements (machines). They are expressed differently via particular properties. The deployment of a software element on a machine (or group of machines) element is expressed by a relationship (UML link) between the two elements.

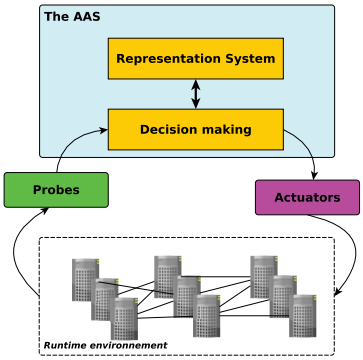

Figure 1. Kephart's AAS Model

(Re)Configuration: to describe reconfigurations programs, TUNe provides a language called Reconfiguration Description Language (RDL) which uses the graphical UML activity diagram. An RDL expression is identified by the name of the event/notification that will trigger its execution. It is a set of sequential or parallel actions whose execution will be carried out on the runtime environment.

This section is not intended to present in detail the TUNe system. For more information, the lecturer can refer to [9].

B. Problematic

As mentioned in the introduction, we are interested in generic AAS. TUNe, as well as Jade, belongs to this category. Although experiments conducted with TUNe validated both the principles of autonomic administration and the component-based approach, they also underlined its limitations. Providing a unique system to manage any type of applications implies that the system must be able to take into account the variation of the administration needs. In addition, one important constraint that the system should be aware of is its usability. It should be useful for users, which often have no knowledge on its implementation. However, the experiments we conducted with several AAS, particularly with TUNe and Jade, showed that the second constraint is not respected. Indeed, when moving from an application area on to another, either the system can not be adapted (e.g. Rainbow with virtualized applications) or its adaptation is very difficult for users (e.g. TUNe with large scale and virtualized applications).

In TUNe (as well as others AASs), the behavior of some administrated elements is hard coded in the system through wrapping). That is the case of machines which are not administrated as software element. But, what about elements such as virtual machines, which are both software and machines? In the Accord system for example, probes elements are administrated differently from others software. The user have to implement them in the AAS. But, what about applications which are built as black box, including

their own monitoring mechanism (e.g. the MySQL server with its MySQL-Safe probe)? In addition, Most of the existing AAS define and impose the administration services they provide. However in some situations these services are not needed or not sufficient. For example, the deployment service is not necessary when administering elements such as printers, which are not deployable (as software).

The causes of these limitations can be summarized as follows:

- Inadequacy of the administration languages makes the AAS not useful in certain situations. It is not relevant to use an unique set of languages for describing all type of applications.
- Strong AAS's implementation hypothesis makes difficult to integrate new administration behaviours for some type of administrated elements.
- Unsuitable AAS design makes difficult to adapt the AAS's services without changing others.

Regarding these problems, the need of a real adaptable and usable AAS remains topical. The next section presents a novel approach to cope these problems.

C. General approach

Our main objective is the construction of an adaptable and usable AAS regardless of the application domains. Administration needs vary according to the application domain or the considered administration facet (deployment, (re)configuration ...). This implies that the construction of a generic AAS should rely neither on any administration language nor on any application domain to avoid complex and difficult to use AAS. To achieve this, we provide on top of the AAS a stack of Model Driven Engineering (MDE) tools allowing to easily define specific administration languages according to the administrated application. Figure 2 summarizes the general approach we use. We try to provide DSL tools which allow administrators to designed their own administration languages in order to capture all the specificities of their administrated applications. These designed languages will therefore be involved in some projection (or model transformation) process by the AAS at the bottom level to perform the autonomic administration.

In summary, administrators will be distinguished into two groups: (AdminGroup1) those who design DSL (step 1) and implement the projection to the AAS (step 2); and (AdminGroup2) those who use DSL (step 3) to describe the environment (software and hardware) to be administrated by the AAS (step 4). The first part of this approach (proof of concept) has been presented in previous works [18][15]. It concerns the implementation of the MDE tools level. The projection of DSLs requires that the AAS be as more flexible and adaptable as possible. The purpose of this paper is the design and the implementation of such an AAS.

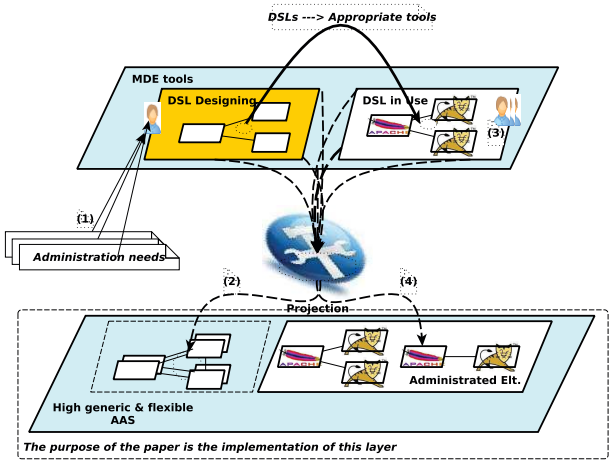

Figure 2. The general approach to implement a flexible and generic AAS

III. Design principles

Our experiments in the implementation of AASs allow us to identify a set of design principles (guidelines) that the development of a higher flexible AAS should respect:

- **Uniformity**: We define a uniform AAS as a system in which differences in role between administered elements are not hard coded in the AAS. In other words, it is an AAS which uses the same internal representation for any administered element regardless of its nature (hardware or software).
- **Adaptability**: An adaptable AAS can evolves according to the needs of administrators. We identify two aspects of adaptability. (1) Adaptation of the AAS's implementation: the capability to modify, replace or add new services without full knowledge of the AAS's implementation. (2) The AAS's ability to adapt itself when administrated elements change. Hence, the AAS will be able to administrate a set of static or evolving elements (by adding software, machines or reconfiguration programs).
- **Interoperability and Collaboration**: The administration of an environment can require the assistance of others AASs or systems. We consider that an AAS is interoperable/collaborative if it is able to exchange informations and administration orders with external systems.
- **Service-oriented component based**: Any service (wrapping, deployment, configuration, etc) of the AAS should be implemented by a well identified component or set of components.

Although all these design principles are orthogonal, the adaptability one is central for the flexibility of the AAS.

1) Uniformity: The recommendation that we propose is: **the AAS should not be aware of the role of the elements it**

administrates. This recommendation is reflected in the AAS by using an uniform representation for any administered element (software, probes, machines, etc). To illustrate the importance of this recommendation, lets us explain two examples which show that an administrated element can play simultaneously several roles. Hence, if the role is hard coded in the AAS, how will elements with several roles be managed?

The execution of a software is a relationship between two entities: the *Execution Unit (EU)* and *Executed Element (EE)*. The *EU* hosts and executes the *EE*. In an administration environment, the hardware always plays the role of EU. Sometimes, software can also act as EU, it is the case with virtual machines. Let us consider the virtual machines example. In one hand, in the relationship [physical machines; VM] the VM is considered as a software from the point of view of the physical machine which hosts the VM. In the other hand, in the relationship [VM; software], the VM acts as an EU from the point of view of the software hosted in the VM. In the same vein, we have J2EE application servers such as JBoss. For the same reasons as the VM, these servers are software. However, their functionality in a J2EE application is to host servlets: they are called servlets containers. They implant all mechanisms for servlets execution and access. For these reasons, they can be considered as EU towards servlets.

2) Adaptability: **AAS Services Adaptability.** Like any computer program, the development of an AAS includes two actors: (1) users (who are administrators in our context) and (2) the AAS's developers. In most cases, the two actors are separated and do not have the same skills. The former has expertise on the application he wants to administrate while the latter master the development techniques of an AAS. On this basis, we define the adaptability of the services of an AAS as its ability to be updated by actors of type (1) without the intervention of actors of type (2). This will allow the AAS to be more generic and flexible. It is one solution among several. let us explain why we use this as a response to implement a generic and flexible AAS.

A generic solution is to provide low-level API to administrators to implement new requirements. This solution is provided by the Jade system [10] with the Fractal API. This solution is intended for warned administrators, what limits its wider usage.

The answer to the limits of the above solution is the provision of higher level tools close to the administrators domain of application. It reduces the AAS's level of genericity by restricting it to a specific application domain. This is the case in the TUNe system which is limited to cluster-type master-slave applications.

The latest solution combines the provision of a high level of abstraction to the genericity of the AAS. It is partially based on the adaptability of the AAS and the providing of higher level tools for expressing administration needs.

Extensibility Since the AAS is not able to predict all achievable administration operations for any type of application, its adaptability should predispose it to integrate new modules for example to take into account new functions or new needs such as the expansion of the environment. For example, allowing migration in virtualized environments in TUNe results in the integration of a new function *migrate*. In summary, we identify three types of extensions that the adaptable AAS should provide:

- The extension of the physical environment: dynamic addition/removal of execution unit (e.g. machines).
- The extension of the software environment: dynamic addition/removal of software. This includes both software which were known in advance by the AAS, and those which description will be integrate in the AAS during its execution. TUNe only provides the former extension capability.
- The extension of reconfiguration policies: it is the ability of the AAS to integrate new administration policies during its execution.

3) Interoperability and Collaboration: The interoperability of a computer system is its ability to interact with others. Under certain conditions, we use the term "collaboration" to refer to interoperability. Indeed, we define collaboration as the connection of several AAS of the same type, while interoperability (more general) brings together several AAS with different types and different designs.

Collaboration in TUNe, as proposed in [19] for administering large-scale applications address a particular problem: scaling of TUNe. It would not apply to the cloud. Indeed, [19] proposes a collaboration between multiple instances of TUNe sharing the administration of large scale application and described by a single administrator. Then all TUNe instances work together to accomplish the administration of the application (which is very large here, thousands of items). Moreover, the communication mechanism between instances are hard coded. However, this solution [19] presents a preliminary step of the collaboration of a AAS. In the case of cloud environment for example, its usage is effective only through collaboration between AAS at the infrastructure level and those at the hosted application level. These AAS have completely different natures. So, they have to be design with interoperability features.

4) Service-oriented component based: contrarily to the TUNe system which uses components only to encapsulate the administrated elements, we propose to apply this approach to the development of the AAS itself. Any service (wrapping, deployment, configuration, etc) of the AAS should be implemented by a well identified component or set of components. This recommendation led us to the service-oriented component architecture shown in Figure 3. Broadly, this architecture is organized around a data structure (called here *RS*, for Representation System) which contains both the administered elements (software and hardware) and the

administration policies. The AAS receives administration requests/orders from external system/human via its component ***External Communicator***. Then, the AAS constructs an internal representation (wrapping) of the administrated environment. This task is done by the ***RS Manager***. After this phase, the AAS is able to carry out others administration tasks which are: the deployment, realized by the ***Deployment Manager***; reconfiguration notifications (from the administrated environment) handling, realized by the ***Event Receiver*** and ***Event Manager***; and performing administration policies (e.g. (re)configuration actions), realized by the ***Policies Manager***. Next sections present in details an implementation of this architecture called TUNeEngine.

IV. TUNeEngine Prototype: A Higher Flexible and Adaptable AAS

This section presents the implementation of each component of the architecture shown in Figure 3, and how they interact to perform the AAS. This implementation is based on the Fractal [16] component model, which was improved in the TUNe system.

A. External Communication

The first step when using an AAS for administering an application is the submission of the administrated environment by an administrator (human) or an external system. This environment includes both the description of the administrated element (hardware and software) and also administration/reconfiguration policies. In the reverse direction, the AAS can initiate communication with an external environment. This is the case for instance when it requests (by collaboration) the services of an external system (e.g. another AAS) to accomplish or complete an administration task. The ***ExternalCommunicator*** component in the architecture implements this service, which represents a dialogue between an external actor (human or computer system) and the AAS. According to the type of the player, the ***ExternalCommunicator*** uses two internal components: the ***Command Line Interface (CLI)*** and the ***Collaborator***. The former deals with human actors, while the latter deals with others systems.

The ***ExternalCommunicator*** is able to handle several types of administration orders. Each of them queries a particular component of the AAS. For example the deployment order will be treated by the ***DeploymentManager*** component. In order to identify the target corresponding component, the ***ExternalCommunicator*** is equipped with an interpreter: ***CmdInterpreter***. As common interpreter, the latter verifies the syntactic and semantic compliance of supplied orders. It then invokes the AAS component which is capable of performing the administration order.

B. Wrapping

Once the administrated environment is transmitted to the AAS, it builds an internal representation of this environment: called the Representation System (***RS***). The latter represents the knowledge base of the AAS. The ***RSManager*** component provides this service. It lies on two components: the ***Parser*** and the ***Wrapper***.

The ***Parser*** identifies in the submitted environment, the list of elements to be administrated by the AAS, their properties, and the relations between them. It can be organized into several ***Parsers*** in order to also take into account elements such as administration policies (or programs). For each identified element, the ***Parser*** asks the ***Wrapper*** to build its internal representation.

Building the representation of an element in the AAS means encapsulate its behaviour in a data structure that will facilitate its administration. This operation is also named *wrapping*. Wrapping elements can be software, machines, links, or elements/actions forming an administration program. Regarding the uniformity criteria we have presented in Section III-1, the ***Wrapper*** is in charge of it. Hence, the implementation of TUNeEngine uses (as we have recommended) the same data structure for encapsulating any type of elements. Finally, the encapsulated elements are kept in the ***RS*** component.

The ***RS*** component plays two roles. Firstly, it represents the data structure which contains the encapsulated elements. Secondly, it provides introspection features to parse its content. It is called upon by other AAS's components to get particular informations on an administrated element (properties, reconfiguration actions, etc). For example, the ***Wrapper*** can request references of two elements when building a binding between them.

C. Deployment

After the wrapping, the deployment phase effectively begins the administration process. It is provided by the ***DeploymentManager*** component, which process is described as follows:

- The choice of the execution support (ES). It is performed by the ***NodeAllocator*** component. It determines the appropriate location for the deployed element. It lays on the ***RS*** to have informations about available ES and then returns one or many ES as needed.
- ES initialization: performed by the ***Deployer***. It initializes the communication between the AAS and the ES: the communication protocol (ssh, rsh, etc) and the authentication informations. These informations are provided by introspecting the ES. The initialization allows the AAS to remotely access the ES.
- Getting and installing binaries of the deployed element. It is provided by the ***BinaryManager***. Firstly, it makes available binary files needed to run the administrated element on the remote ES (e.g. installing packages in Linux). Then, it organizes the files according to the installation tree required by the administrated element.

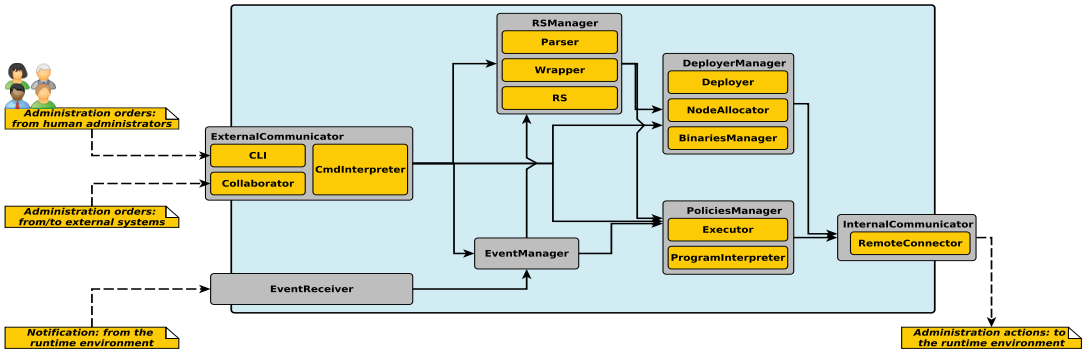

Figure 3. Architecture of TUNeEngine

Conversely, note that the components presented above also perform the undeployment operation. In this case, the ***NodeAllocator*** releases the ES after it has been cleaned by the ***BinaryManager***.

D. Configuration and Startup

The configuration and the startup phases come after the deployment. Because they are similar in nature (execution of the startup/configuration programs), the AAS uses the same component (***PoliciesManager***) to achieve them. These phases start when the AAS receives an administration order from the ***ExternalCommunicator***. The execution of an administration policy (which is a program, a set of actions) is performed into two steps:

- interpretation of the configuration/startup program: conducted by the ***ProgramInterpreter***. Remember that programs (as well as administrated elements) reside in the ***RS***. The ***ProgramInterpreter*** parses the program in order to identifies the list of actions to perform.
- execution of actions identified in the previous step: realized by the ***Executor***. For each action, the ***Executor*** introspects the ***RS*** in order to identify elements referenced in the action. It then lays on the ***RemoteConnector*** component to remotely execute the action on the runtime element (software or hardware).

E. Reconfiguration

After the administrated elements has been started, then comes their administration which consists in monitoring them in order to inform the AAS in case of particular changes. The realization of this task does not belong to the AAS. It is the responsibility of the administrator to define among its administrated elements, some particular ones which play the role of probe. Indeed, in order to provide a generic AAS, we make no difference between probes and other administrated elements (all are seen as a black box). However, the AAS provides the mechanism to realize communication between administrated elements and itself (through out notifications).

For reconfigurations, the communication is initiated by the administered element, from its ES to the machine which runs the AAS (the administration machine). The ***EventDriver*** component of the AAS is used by the administrated element to emit notifications (also called events) from the ES to the administration machine. Events are received to the administration machine by the ***EventReceiver*** which forwards it to the ***EventManager***. The latter decides if the treatment of the event is necessary. This decision depends on the state of elements in the ***RS***. If the treatment is considered, it chooses the appropriate reconfiguration program which execution will resolve the reported problem. Finally the ***PoliciesManager*** performs the execution of the program, as we have described in the previous section.

V. Use case: Autonomic management of Cloud platforms

To illustrate the adaptability of our prototype TUNeEngine, we use it to administrate a cloud computing [20] platforms as well as the hosted applications.

A. Evaluation Context

A cloud computing platform is a hosting center which objective is to share the same infrastructure to several applications belonging to distinct users, who are billed in a pay-as-you go model. It is generally based on the virtualization [12] technology (capability to run simultaneously several OS, called virtual machines, on the same machine) which facilitates and improves the cloud provider benefits (by increasing the hosting capacity). We choose this use case because it brings variable administration needs we can meet in several types of applications. This section presents them and how the TUNeEngine prototype is adapted to address them.

Figure 4 presents a simplify architecture of a cloud computing infrastructure where AASs are needed. It can be interpreted as follow:

- At the infrastructure level: we have one or several AASs instances which manage VMs runtime, VM file

systems, VM network, resource allocation to VM and monitoring.
- At the application level: we have one AAS instance per application. They are able to interact with the infrastructure's AAS instances in order to start or stop VMs. For this evaluation, we suppose that applications which are hosted in the cloud are multi-tier type such as JEE.

B. Administration needs and Adaptation in TUNeEngine

Administration Need 0:
Any component of the Cloud architecture should be considered, either as an administrated element (e.g. VM and VM file systems) or as a reconfiguration program (components which manage VM file system storage, network initialization, and VM life cycle). This need is taken into account in the AAS by the ***RS*** component.

Administration Need 1:
The communication between the cloud and its clients should be done in a comprehensive way. A widely used API to achieve this is the REST protocol (e.g euca2ools [21]). This is implemented in the AAS by adapting the ***Collaboration*** component.

Administration Need 2:
The administration of the cloud infrastructure can require several instances of AAS_{cloud} when the infrastructure is very large (thousands of machines). Indeed, a single instance can leads to a bottleneck. Therefore, the instances should be able to collaborate with each other in order to achieve the administration of the overall infrastructure. This requires some adaptations of the AAS: sharing the ***RS*** data structure between all AAS_{cloud}; adapting the ***Collaboration*** component for collaboration, and adapt the ***Event Driver*** to identify the appropriate AAS_{cloud} instance when an event is emitted from the runtime.

Administration Need 3:
By definition, the cloud should be able to start new VMs when an application requests for new resources. Since the AAS_{cloud} keeps a representation of each element it manages, archiving the addition/removal of new virtual machines implies that the AAS_{cloud} should be able to dynamically integrate/remove elements to its ***RS*** component. This is not the case when administering static environments. This need is also presented at the AAS_{app}. Indeed, exploiting the cloud advantages such as the facility to allocate or free resources in terms of minutes (instead of days in an IT company), the cloud customers generally implements their application with elastic behaviour. In order word, they start their application with minimal resources and adapt them (by adding/removing resources) according to the workload. For example, in a JEE e-commerce application, the number of data base server increases/decreases according to the workload of the number of Internet traffics on the application.

Administration Need 4:
Traditionally, applications run on physical machines. In a virtualized cloud, they run on VMs. However, VMs are also applications since they run on physical machines. In summary, we have a stack of runtime environments: cloud customers applications on VMs, and VMs on physical machines. This reflects in the AAS_{cloud} by its ability to consider a VM both as a software element and as an execution support. This requires the adaptation of both the ***Wrapper*** (for the encapsulation of VMs) and the ***Deployer*** (for the deployment of VMs) components.

Administration Need 5:
Finally, the administration of the cloud environment requires the integration of several reconfiguration policies. These are defined as programs and integrated in the ***RS*** component. Here is a non exhaustive list of them:

- Elasticity: the addition/removal of components both on AAS_{cloud} and AAS_{app} instances.
- Machines allocation: what is the best machine to host VMs while minimizing the total number of machines used in the cloud. This is often called VM placement.
- VMs migration: increase the hosting capacity of the cloud is one of the most import objective of the cloud provider. This is generally improved using live VM migration [12] to group applications on a minimum number of machines.

VI. Related Works

Among existing AAS, few has the vocation to administrate several type of applications. Rainbow [8] is one of the first AAS in this category. Its architecture is organized into two parts: the first part implements the basic functionality of self-administration while the second part implements the services which is adaptable. Except for the lack of the deployment and the collaboration services, the architecture of Rainbow is close to the architecture we present in this paper. Rainbow provides two languages which are hard coded and not adaptable. It organizes the ***RS*** into two categories: one for software and the other for machines elements. Moreover , it hard codes the difference between machines, software and probes elements. The latter are not considered as manageable element. There is no uniformity in Rainbow.

Accord [14] is a generic AAS in the same vein as TUNe. It considers administrated elements as black boxes. However, it does not take into account the administration of machines element. Similarly to Rainbow, it makes a difference between software that perform business functions and probes acting as monitor. The development of Accord does not follow a component-based approach. There is no way to change/replace/add new features by an administrator.

Unity [13] is the first AAS developed after the introduction of the autonomic administration principles in 2003. Similarly to what we proposed in this paper, Unity uses

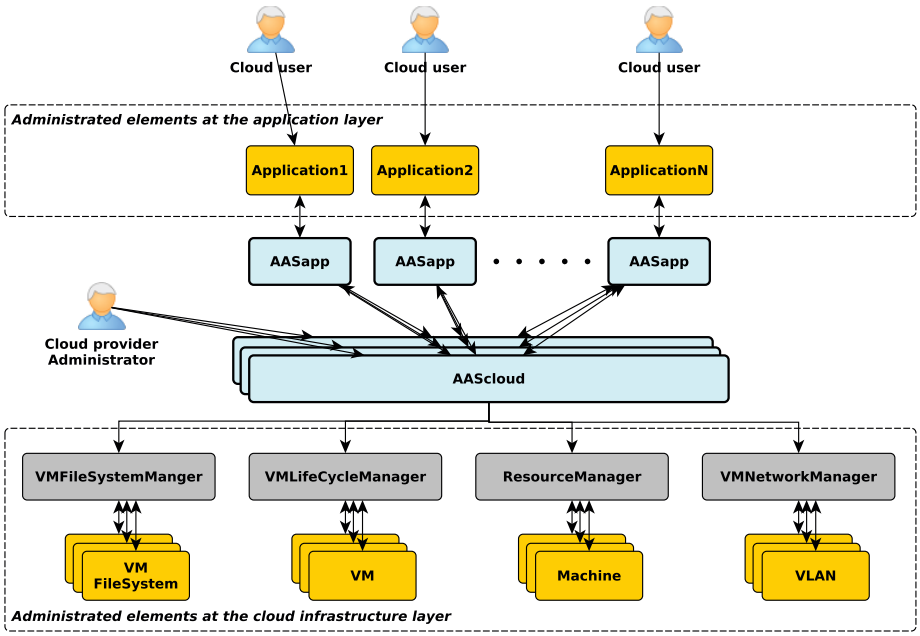

Figure 4. The evaluation use case: a simplify organization of a cloud platform

a component model for both its implementation and the encapsulation of administered element. It defines a particular data structure, which is hard coded, for each type of element it administrates. A well known behaviour is associated to each of them. Although Unity allows dynamic integration of new elements at runtime (reconfiguration programs, machines, or software), it does not provide any way to adapt its components.

These last years have seen some research work on the implementation of adaptable AASs. As we propose in this paper, most of them are based on component model and Model Driven Engineering. [22] describes a general approach to generate specific AASs from DSL which describes an application domain. It only focuses on the description of the model-driven part while we address in this paper the implementation of the adaptable AAS. Ceylon project [23] attempts to build a generic and flexible AAS. It proposes a service-oriented component approach to do that. It focuses on the communication workflow in the AAS. No architecture of the Ceylon system is provided. [24] completes the work proposed by Ceylon. It defines some design patterns for the implementation of a generic AAS. Most of them are presented in our work (e.g. AAS extensibility). [25] proposes a framework which is similar to Ceylon.

VII. Conclusion and Perspectives

This work is part of the general approach we adopted in [18] for the construction of a generic and useful AAS. This approach includes two stages: (1) use of the model-driven technology (through DSLs) to raise the AAS's level of abstraction and (2) lay on a high flexible and adaptable AAS to support any administration needs. After the presentation of the first stage in a previous work [15], this paper has focused on the second stage. We proposed some design principles that the development of such AASs should follow. These are uniformity (the behaviour of a type of administrated elements should not be hard-coded in the AAS), adaptability (any service of the AAS is adaptable without knowledge on the entire implementation of the AAS), and collaboration/interoperability (the AAS is able to communicate with external systems). We described the implementation of a prototype, TUNeEngine, based on a service-oriented component approach. This prototype has been evaluated through the administration of a virtualized cloud computing platform, including the applications it hosts. As future work, we plan to combine the two stages of the approach in order to provide the full generic and useful AAS we claim for.

Acknowledgment

The work reported in this paper benefited from the support of the French National Research Agency through project SelfXL (ANR-08-SEGI-017-04).

References

[1] J. O. Kephart and D. M. Chess. The vision of autonomic computing. In *IEEE Computer Magazine, 36(1)*, 2003.

[2] Paul Horn. Autonomic computing: IBM's Perspective on the State of Information Technology. posted at 2007-07-30 09:52:32 by IBM.

[3] Richard Murch. Autonomic Computing. published by IBM press, 2004.

[4] Mohammed Toure, Girma Berhe, Patricia Stolf, Laurent Broto, Noel Depalma, and Daniel Hagimont. Autonomic Management for Grid Applications. Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based, pp. 79-86, 2008.

[5] Julie A. McCann, Gawesh Jawaheer, and Linxue Sun. Patia: Adaptive Distributed Webserver (A Position Paper). Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems, 2003.

[6] Julie A. McCann, Gawesh Jawaheer, and Linxue Sun. Patia: Adaptive Distributed Webserver (A Position Paper). Proceedings of the The Sixth International Symposium on Autonomous Decentralized Systems, 2003.

[7] Benoit Claudel, Guillaume Huard, and Olivier Richard, *Taktuk, adaptive deployment of remote executions*, in the Proceedings of the ACM international symposium on High performance distributed computing, pp. 91-100, New York, NY, USA, 2009.

[8] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste, *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*, in the IEEE Computer, Vol. 37, Issue 10, pp. 46-54, 2004.

[9] Laurent Broto, Daniel Hagimont, Patricia Stolf, Noel Depalma, and Suzy Temate, *Autonomic management policy specification in Tune*, in Proceedings of the ACM symposium on Applied computing, pp. 1658-1663, Fortaleza, Ceara, Brazil, 2008.

[10] Sara Bouchenak, Fabienne Boyer, Sacha Krakowiak, Daniel Hagimont, Adrian Mos, Stefani Jean-Bernard, Noel de Palma, and Vivien Quema, *Architecture-Based Autonomous Repair Management: An Application to J2EE Clusters*, in Proceedings of the Symposium on Reliable Distributed Systems, pp. 13-24, Orlando, FL, USA, 2005.

[11] Alain Tchana, Suzy Temate, Laurent Broto, Daniel Hagimont, *Autonomic resource allocation in a J2EE cluster*, in EEE International Conference. Utility and Cloud Computing, Chennai, Inde, 2010.

[12] Paul Braham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Haris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, *Xen and the art of virtualization*, in Proceedings of the ACM symposium on Operating systems principles, pp. 164-177, New York, USA, 2003.

[13] David M. Chess, Alla Segal, Ian Whalley, and Steve R. White, *Unity: Experiences with a Prototype Autonomic Computing System*, in Proceedings of the International Conference on Autonomic Computing, pp. 140-147, New York, USA, 2004.

[14] Hua Liu, Manish Parashar, *Accord: A Programming Framework for Autonomic Applications*, in IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews, Vol. 36, Issue 3, pp. 341-352, 2006.

[15] Suzy Temate, Laurent Broto, Alain Tchana, and Daniel Hagimont, *A High Level Approach for Generating Model's Graphical Editors*, in Proceedings of the International Conference on Information Technology: New Generations, pp. 743-749, Las Vegas, Nevada, USA, 2011.

[16] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani, *The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems*, in Softw. Pract. Exper., vol. 36, ISSN: 0038-0644, pp. 1257-1284, 2006.

[17] Object Management Group, Inc., *Unified Modeling Language (UML) 2.1.2 Superstructure*, in the Final Adopted Specification, 2007.

[18] Benoît Combemale, Laurent Broto, Xavier Crégut, Michel J. Daydé, and Daniel Hagimont, *Autonomic Management Policy Specification: From UML to DSML*, in Proceedings of the Model Driven Engineering Languages and Systems, pp. 584-599, Toulouse, France, 2008.

[19] Mahamadou Toure, Patricia Stolf, Daniel Hagimont, Laurent Broto, *Large Scale Deployment*, in Proceedings of the International Conference on Autonomic and Autonomous Systems, pp. 78-83, Cancun, Mexico, 2010.

[20] Rajkumar Buyya, Chee Shin Yeo, and Srikumar Venugopal, *Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities*, in Proceedings of the International Conference on High Performance Computing and Communications, pp. 5-13, DaLian, China, 2008.

[21] Euca2ools User Guide, in *http://open.eucalyptus.com/wiki/Euca2oolsGuide*, visited April 2012.

[22] Holger Kasinger and Bernhard Bauer, *Towards a model-driven software engineering methodology for organic computing systems*, in Proceedings of the 4th IASTED International Conference on Computational Intelligence, IASTED/ACTA Press, pp. 141-146, Calgary, Alberta, Canada, July 2005.

[23] Yoann Maurel, Ada Diaconescu, and Philippe Lalanda, *CEYLON : A service-oriented framework for building autonomic managers*, in Proceedings of IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems, University of Oxford, England, March 2010.

[24] Sylvain Frey, Ada Diaconescu, and Isabelle Demeure, *Architectural Integration Patterns for Autonomic Management Systems*, in Proceedings of IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems, Novi Sad, Serbia, April 2012.

[25] Radu Calinescu, *Implementation of a Generic Autonomic Framework*, in Proceedings of the International Conference on Autonomic and Autonomous Systems, Gosier, Guadeloupe, March 2008.